Loo.py: A Loop Generation Tool for CPUs and GPUs

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

March 1, 2012
Thanks

- Tim Warburton (Rice)
- Jan Hesthaven (Brown)
- Leslie Greengard (NYU)
- Hendrik Riedmann (Stuttgart/Brown)
- PyOpenCL, PyCUDA contributors
- AMD, Nvidia
Outline

1. Introduction
2. A Different Way of Writing Compute Codes
3. Code Generation, Generally
4. Conclusions
Outline

1. Introduction
   - Discontinuous Galerkin Methods
   - GPU-DG: Challenges

2. A Different Way of Writing Compute Codes

3. Code Generation, Generally

4. Conclusions
Outline

1. Introduction
   - Discontinuous Galerkin Methods
   - GPU-DG: Challenges

2. A Different Way of Writing Compute Codes

3. Code Generation, Generally

4. Conclusions
Discontinuous Galerkin Method

Let $\Omega := \bigcup_i D_k \subset \mathbb{R}^d$.

Goal

Solve a conservation law on $\Omega$:

$$u_t + \nabla \cdot F(u) = 0$$

Example

Maxwell’s Equations: EM field: $E(x, t), H(x, t)$ on $\Omega$ governed by

$$\partial_t E - \frac{1}{\varepsilon} \nabla \times H = -\frac{j}{\varepsilon},$$

$$\nabla \cdot E = \frac{\rho}{\varepsilon},$$

$$\partial_t H + \frac{1}{\mu} \nabla \times E = 0,$$

$$\nabla \cdot H = 0.$$
Discontinuous Galerkin Method

Multiply by test function, integrate by parts:

\[ 0 = \int_{D_k} u_t \varphi + \left[ \nabla \cdot F(u) \right] \varphi \, dx \]
\[ = \int_{D_k} u_t \varphi - F(u) \cdot \nabla \varphi \, dx + \int_{\partial D_k} (\hat{n} \cdot F) \varphi \, dS_x, \]

Substitute in basis functions, introduce elementwise stiffness, mass, and surface mass matrices matrices \( S, M, M_A \):

\[ \partial_t u^k = - \sum_{\nu} D^{\partial \nu, k} [F(u^k)] + L^k \left[ \hat{n} \cdot F - (\hat{n} \cdot F)^* \right]_{A \subset \partial D_k}. \]

For straight-sided simplicial elements:
Reduce \( D^{\partial \nu} \) and \( L \) to reference matrices.
Outline

1. Introduction
   - Discontinuous Galerkin Methods
   - GPU-DG: Challenges

2. A Different Way of Writing Compute Codes

3. Code Generation, Generally

4. Conclusions
Element-Local Operations: Differentiation

Local Templated Derivative Matrices

Field Data

Geometric Factors

$N_p$

$K$

$Loo.py$: A Loop Generation Tool
Loop Slicing for element-local parts of GPU DG

Per Block: $K_L$ element-local mat.mult. + matrix load

Question: How should one assign work to threads?
Loop Slicing for element-local parts of GPU DG

Per Block: $K_L$ element-local mat.mult. + matrix load

Question: How should one assign work to threads?

$w_s$: in sequence

$w_i$: “inline-parallel”

$w_p$: in parallel

(amortize preparation)  (exploit register space)
DG on GPUs: Implementation Choices

- Many difficult questions
- Insufficient heuristics
- Answers are hardware-specific and have no lasting value
DG on GPUs: Implementation Choices

- Many difficult questions
- Insufficient heuristics
- Answers are hardware-specific and have no lasting value

Tune automatically at computation time, cache tuning results.

- Decrease reliance on knowledge of hardware internals
- Shift emphasis from tuning results to tuning ideas
Outline

1. Introduction

2. A Different Way of Writing Compute Codes
   - RTCG Infrastructure
   - RTCG for DG

3. Code Generation, Generally

4. Conclusions
Outline

1. Introduction

2. A Different Way of Writing Compute Codes
   - RTCG Infrastructure
   - RTCG for DG

3. Code Generation, Generally

4. Conclusions
Metaprogramming

In compute scripting, kernel code does not need to be a compile-time constant.
Metaprogramming

In *compute scripting*, kernel code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
In *compute scripting*, kernel code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
In *compute scripting*, kernel code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
Metaprogramming

In *compute scripting*, kernel code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
In *compute scripting*, kernel code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)

In *compute scripting*, kernel code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
In *compute scripting*, kernel code does *not* need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
Metaprogramming

In compute scripting, kernel code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In compute scripting, kernel code does not need to be a compile-time constant.

(Key: Code is data—it *wants* to be reasoned about at run time)
PyOpenCL, PyCUDA: Vital Information

- [link](http://mathema.tician.de/software/pyopencl) (or /pycuda)
- Complete documentation
- MIT License
- Arrays, Elementwise op., Reduction, Scan
- Compiler Cache, RAII, Error checking
- Requires: numpy, Python 2.4+ (Win/OS X/Linux)
- Community: downloaded 80,000+ times, available in all major Linux distributions, mailing list, wiki, add-on packages (FFT, scikits.cuda, ...)

OpenCL

Andreas Klöckner

Loo.py: A Loop Generation Tool
Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
  \[\Rightarrow\text{complement each other}\]

- CPU: largely restricted to control tasks (\(\sim 1000/\text{sec}\))
  - Scripting fast enough
import pyopencl as cl, numpy

a = numpy.random.rand(256**3).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_dev)

prg = cl.Program(ctx, ""
    __kernel void twice(__global float *a)
    { a[ get_local_id(0)+ get_local_size(0)*get_group_id(0)] *= 2; }
""").build()

prg.twice(queue, a.shape, (256,), a_dev)
A taste of PyOpenCL

```python
import pyopencl as cl, numpy

a = numpy.random.rand(256**3).astype(numpy.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(queue, a_dev)

prg = cl.Program(ctx, "
__kernel void twice(__global float *a)
{ a[ get_local_id(0)+ get_local_size(0)*get_group_id(0) ] *= 2; }
"").build()

prg.twice(queue, a.shape, (256,), a_dev)
```

Compute kernel
### Outline

1. Introduction

2. A Different Way of Writing Compute Codes
   - RTCG Infrastructure
   - RTCG for DG

3. Code Generation, Generally

4. Conclusions
Metaprogramming for GPU-DG

- Specialize code for user-given problem:
  - Flux Terms

- Automated Tuning:
  - Memory layout
  - Loop slicing
  - Gather granularity

- Constants instead of variables:
  - Dimensionality
  - Polynomial degree
  - Element properties
  - Matrix sizes

- Loop Unrolling
Loop Slicing for Differentiation

Local differentiation, matrix-in-shared, order 4, with microblocking. Point size denotes $w_i \in \{1, \ldots, 4\}$.
Nvidia GTX280 vs. single core of Intel Core 2 Duo E8400

![Graph showing GFlops/s vs. Polynomial Order N for GPU and CPU]
Memory Bandwidth on a GTX 280

![Graph showing memory bandwidth for different polynomial orders and operations.](Image)

- **Polynomial Order** $N$: 20, 40, 60, 80, 100, 120, 140, 160, 180, 200
- **Global Memory Bandwidth [GB/s]**: Gather, Lift, Diff, Assy., Peak

Andreas Klöckner  
Loo.py: A Loop Generation Tool
Outline

1. Introduction

2. A Different Way of Writing Compute Codes

3. Code Generation, Generally
   - Loo.py

4. Conclusions
Outline

1. Introduction

2. A Different Way of Writing Compute Codes

3. Code Generation, Generally
   - Loo.py

4. Conclusions
Automating GPU Programming

GPU/OpenCL programming can be a time-consuming trial-and-error process.

Obvious idea: Let the computer do it. How?

- One way: “Smart” compiler, “dumb” developer
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: “Smart” developer, “dumb” compiler
  - Error-prone
  - Expensive in developer time
  - User can use manual/automatic tuning
Automating GPU Programming

GPU/OpenCL programming can be a time-consuming trial-and-error process.

Obvious idea: Let the computer do it. How?

So compromise! Following: our idea of a compromise.

- Heuristics are fragile
- Another way: “Smart” developer, “dumb” compiler
  - Error-prone
  - Expensive in developer time
  - User can use manual/automatic tuning
Idea: A library of code transformations

- Start with a simple, mostly mathematical statement of the desired operation
- “Push a few buttons” to uniquely specify program for one target device.
- Strongly separate these two parts
  - Operation spec stays the same
  - “Button combinations” can change by target device

Non-Goals:

- Anything that requires heuristics/“intelligence”
- Or autotuning
Setting the Stage

Idea: A library of code transformations

- Start with a simple, mostly mathematical statement of the desired operation
- “Push a few buttons” to uniquely specify program for one target device.
- Strongly separate these two parts
  - Operation spec stays the same
  - “Button combinations” can change by target device

Non-Goals:

- Anything that requires heuristics/“intelligence” or autotuning

Loo.py is infrastructure.

Auto-tuners and domain-specific libraries are “above” loopy conceptually.

Specifically: Loo.py is intended to make kernel-specific autotuners very cheap to build.
So, GPU-OpenMP?

- Some overlap with directive-based approaches
  - But not really
- “Directives” live in Python
  - No reason directives should be static
  - Can write functions to capture generic transformations
  - Loop over parameter: Instant auto-tuner
- Transformations, not directives!
  - LoopKernel data type is a Python object
  - `some_transform(knl)` gives you a new `LoopKernel`
- Unique specification of output C code
Components of a loopy program:

- A loop domain, given as a set of constraints:
  \[ \{[i,j,k]: \ 0 \leq i,j,k < 20\} \] (convex)

  Call each axis an iname.

- A set of assignment instructions, each with
  - LHS expression, RHS expression
  - Set of inames within which it is run
  - A label

- A set of arguments (nd arrays or scalars)

- An target OpenCL device
Run-time parameters

The loop domain may be parametrized:

\[ [n] \rightarrow \{ [i,j,k]: 0 \leq i,j,k < n \} \]

- Flexible
  - Some bounds fixed at generation time
  - Others specified at run time
- Each domain parameter must also occur as a scalar argument
- Also possible to encode *assumptions* on parameters:
  Example: \( n \geq 1 \)
Example: Rank-one matrix generation

```python
import loopy as lp

knl = lp.make_kernel(cl.device,
    "[n] -> {[i,j]: 0<=i,j<n}",
    ["label: c[i, j] = a[i]*b[j]"
    ],
    [lp.ArrayArg("a", dtype, shape="n,", order=order),
    lp.ArrayArg("b", dtype, shape="n,", order=order),
    lp.ArrayArg("c", dtype, shape="n, n", order=order),
    lp.ScalarArg("n", np.int32, approximately=1000),
    ],
    name="rank_one", assumptions="n >= 16")
```
Example: Rank-one matrix generation

```c
__kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
rank_one(
    __global float const * restrict a,
    __global float const * restrict b,
    __global float * restrict c, int const n)
{
    for (int j = 0; n + -1 - j >= 0; ++j)
        for (int i = 0; n + -1 - i >= 0; ++i)
            c[n*i + j] = b[j]*a[i];
}
```

Actually generates two kernels—one i-in-j, one the other way around.
Example: Rank-one matrix generation

```cpp
knl = lp.split_dimension(knl, "i", 16,
    outer_tag="g.0", inner_tag="l.0")
```
```
knl = lp.split_dimension(knl, "j", 16,
    outer_tag="g.1", inner_tag="l.1")
```

```cpp
__kernel void __attribute__((reqd_work_group_size(16, 16, 1)))
rank_one (...) {

    if (n + gid(0)*(-16) + -1 - lid(0) >= 0
        && n + gid(1)*(-16) + -1 - lid(1) >= 0
    )
        c[n*(lid(0) + gid(0)*16) + lid(1) + gid(1)*16] =
        b[lid(1) + gid(1)*16]*a[lid(0) + gid(0)*16];
}
```
Example: Rank-one matrix generation

```python
knl = lp.add_prefetch(knl, "a")
knl = lp.add_prefetch(knl, "b")
```

```python
{
    float  fetch_b;
    float  fetch_a;

    if (
        n + gid(0)*(-16) + -1 - lid(0) >= 0
        && n + gid(1)*(-16) + -1 - lid(1) >= 0
    )
    {
        fetch_a = a[lid(0) + gid(0)*16];
        fetch_b = b[lid(1) + gid(1)*16];
        c[n*(lid(0) + gid(0)*16) + lid(1) + gid(1)*16] = fetch_b*fetch_a ;
    }
}
```
Example: Rank-one matrix generation

```python
knl = lp.add_prefetch(knl, "a", ["i_inner"])
knl = lp.add_prefetch(knl, "b", ["j_inner"])

{
  __local  float  fetch_b [16];
  __local  float  fetch_a [16];

  if (n + gid(0)*(-16) + -1 - lid(0) >= 0)
    fetch_a [lid (0)] = a[lid (0) + gid(0)*16];
  if (n + gid(1)*(-16) + -1 - lid(0) >= 0)
    fetch_b [lid (0)] = b[lid (0) + gid(1)*16];
  barrier (CLK_LOCAL_MEM_FENCE) /* dependency: fetch_a */;
  if (n + gid(0)*(-16) + -1 - lid(0) >= 0
      && n + gid(1)*(-16) + -1 - lid(1) >= 0)
    c[n*(lid (0) + gid(0)*16) + lid (1) + gid(1)*16]
      = fetch_b[lid (1)]*fetch_a[lid (0)];
}
```
Example: Rank-one matrix generation

```python
knl = lp.split_dimension(knl, "i", 256,
                        outer_tag="g.0")
knl = lp.split_dimension(knl, "j", 256,
                        outer_tag="g.1")
knl = lp.add_prefetch(knl, "a", ["i_inner"], default_tag=None)
knl = lp.add_prefetch(knl, "b", ["j_inner"], default_tag=None)
knl = lp.split_dimension(knl, "i_inner", 16,
                        inner_tag="l.0")
knl = lp.split_dimension(knl, "j_inner", 16,
                        inner_tag="l.1")
knl = lp.split_dimension(knl, "j_inner_fetch_b", 16,
                        outer_tag="l.1", inner_tag="l.0")
knl = lp.split_dimension(knl, "i_inner_fetch_a", 16,
                        outer_tag="l.1", inner_tag="l.0")
```
Example: Rank-one matrix generation

```python
{
    __local  float  fetch_b [256];
    __local  float  fetch_a [256];

    if (n + lid(1)*(-16) + gid(0)*(-256) + -1 - lid(0) >= 0)
        fetch_a [ lid (1)*16 + lid (0)] = a[lid (1)*16 + lid (0) + gid(0)*256];
    if (n + lid(1)*(-16) + gid(1)*(-256) + -1 - lid(0) >= 0)
        fetch_b [ lid (1)*16 + lid (0)] = b[lid (1)*16 + lid (0) + gid(1)*256];
    barrier (CLK_LOCAL_MEM_FENCE) /* dependency: fetch_a */;

    for (int  i_inner_outer  = 0; 15 - i_inner_outer  >= 0; ++i_inner_outer)
        if (n + i_inner_outer *(-16) + gid(0)*(-256) + -1 - lid(0) >= 0)
            for (int  j_inner_outer  = 0; 15 - j_inner_outer  >= 0; ++j_inner_outer)
                if (n + j_inner_outer *(-16) + gid(1)*(-256) + -1 - lid(1) >= 0)
                    c[n*( lid (0) + i_inner_outer *16 + gid(0)*256)
                        + lid (1) + j_inner_outer *16 + gid(1)*256] =
                        fetch_b [ lid (1) + j_inner_outer *16]*fetch_a [ lid (0)
                        + i_inner_outer *16];
}
```
Example: Rank-one matrix generation

```python
knl = lp.split_dimension(knl, "i", 256, outer_tag="g.0", slabs=(0, 1))
knl = lp.split_dimension(knl, "j", 256, outer_tag="g.1", slabs=(0, 1))
```
Example: Rank-one matrix generation

```c
/* bulk slab for 'i_outer' */
/* bulk slab for 'j_outer' */
if (  
    -3 + ((n + 255)/256) - gid(0) >= 0
    && -3 + ((n + 255)/256) - gid(1) >= 0
)
{
    fetch_a [lid (1)*16 + lid (0)] = a[lid (1)*16 + lid (0) + gid(0)*256];
    fetch_b [lid (1)*16 + lid (0)] = b[lid (1)*16 + lid (0) + gid(1)*256];
    barrier (CLK_LOCAL_MEM_FENCE) /* dependency: fetch_a */;
    for (int i_inner_outer = 0; 15 - i_inner_outer >= 0; ++i_inner_outer)
        for (int j_inner_outer = 0; 15 - j_inner_outer >= 0; ++j_inner_outer)
            c[n*(lid (0) + i_inner_outer *16 + gid(0)*256)
                + lid(1) + j_inner_outer *16 + gid(1)*256] =
                fetch_b [lid (1) + j_inner_outer *16]*fetch_a [lid (0) + i_inner_outer *16];
}
/* final slab for 'j_outer' */
if (  
...

(73 lines)
```
Example: Matrix multiplication

```python
knl = lp.make_kernel(ctx.devices[0],
    "{[i,j,k]: 0<=i,j,k<%d}" % n,
    [  
        "c[i, j] = sum_float32(k, a[i, k]*b[k, j])"
    ],
    [
        lp.ArrayArg("a", dtype, shape=(n, n), order=order),
        lp.ArrayArg("b", dtype, shape=(n, n), order=order),
        lp.ArrayArg("c", dtype, shape=(n, n), order=order),
    ],
    name="matmul")
```
Example: Matrix multiplication

```c
__kernel void __attribute__((reqd_work_group_size(1, 1, 1)))
matmul(
    __global float const * restrict a,
    __global float const * restrict b,
    __global float * restrict c)
{
    float acc;

    for (int j = 0; 159 - j >= 0; ++j)
        for (int i = 0; 159 - i >= 0; ++i)
            {
                acc = 0;
                for (int k = 0; 159 - k >= 0; ++k)
                    acc = b[k*160 + j]*a[k + i*160] + acc;
                c[j + i*160] = acc;
            }
}
```
Example: Matrix multiplication

```python
knl = lp.split_dimension(knl, "i", 16,
    outer_tag="g.0", inner_tag="l.1")
knl = lp.split_dimension(knl, "j", 16,
    outer_tag="g.1", inner_tag="l.0")
knl = lp.split_dimension(knl, "k", 16)
knl = lp.add_prefetch(knl, 'a', ['k_inner', 'i_inner'])
knl = lp.add_prefetch(knl, 'b', ['j_inner', 'k_inner'])
```
Example: Matrix multiplication

```c
__kernel void __attribute__((reqd_work_group_size(16, 16, 1))) ...
{
    __local float fetch_b [16][16];
    float acc;
    __local float fetch_a [16][16];

    acc = 0;
    for (int k_outer = 0; 9 - k_outer >= 0; ++k_outer)
    {
        barrier (CLK_LOCAL_MEM_FENCE) /* pre-barrier: fetch_b */;
        fetch_b[ lid (0)][ lid (1)] = b[ lid (0) + gid(1)*16 + 160*(lid(1) + k_outer*16)];
        fetch_a[ lid (0)][ lid (1)] = a[ lid (0) + k_outer*16 + 160*(lid(1) + gid(0)*16)];
        barrier (CLK_LOCAL_MEM_FENCE) /* dependency: fetch_a */;
        for (int k_inner = 0; 15 - k_inner >= 0; ++k_inner)
        {
            acc = fetch_b[ lid (0)][ k_inner]*fetch_a[ k_inner ][ lid (1)] + acc;
            c[ lid (0) + gid(1)*16 + 160*(lid(1) + gid(0)*16)] = acc;
        }
    }
}
```

Andreas Klöckner

Loo.py: A Loop Generation Tool
Example: Matrix multiplication

```python
knl = lp.split_dimension(knl, "k_inner", 4, inner_tag="unr")

barrier(CLK_LOCAL_MEM_FENCE)  # dependency: fetch_a */;
for (k_inner_outer = 0; 3 - k_inner_outer >= 0; ++k_inner_outer)
{
    acc = fetch_b[ lid (0)][ k_inner_outer *4 + 0]
    *fetch_a [ k_inner_outer *4 + 0][ lid (1)]  + acc;
    acc = fetch_b[ lid (0)][ k_inner_outer *4 + 1]
    *fetch_a [ k_inner_outer *4 + 1][ lid (1)]  + acc;
    acc = fetch_b[ lid (0)][ k_inner_outer *4 + 2]
    *fetch_a [ k_inner_outer *4 + 2][ lid (1)]  + acc;
    acc = fetch_b[ lid (0)][ k_inner_outer *4 + 3]
    *fetch_a [ k_inner_outer *4 + 3][ lid (1)]  + acc;
}
c[lid(0) gid(1)*16 + 160*(lid(1) + gid(0)*16)] = acc;
```
Example: Matrix multiplication

```python
knl = lp.split_dimension(knl, "i", 16,
                          outer_tag=None, inner_tag="l.1")

knl = lp.split_dimension(knl, "j", 16,
                          outer_tag="g.1", inner_tag="l.0")

knl = lp.split_dimension(knl, "k", 16)

knl = lp.split_dimension(knl, "i_outer", 4,
                          inner_tag="ilp", outer_tag="g.0")

knl = lp.add_prefetch(knl, 'a',
                      ['k_inner', 'i_inner', 'i_outer_inner'])

knl = lp.add_prefetch(knl, 'b', ['j_inner', 'k_inner'])

knl = lp.tag_dimensions(knl, {'i_outer_inner_fetch_a': None})
```
Example: Matrix multiplication

```python
for (int k_outer = 0; 19 - k_outer >= 0; ++k_outer)
{
    barrier (CLK_LOCAL_MEM_FENCE) /* pre–barrier: fetch_b */;
    fetch_b [ lid(0)][ lid(1)] = b[lid(0) + gid(1)*16 + 320*(lid(1) + k_outer*16)];
    for (int i_outer_inner_fetch_a = 0; 3 - i_outer_inner_fetch_a >= 0;
         ++i_outer_inner_fetch_a )
        fetch_a [ lid(0)][ lid(1)][ i_outer_inner_fetch_a ] =
        a[lid(0) + k_outer*16 + 320*(lid(1)
          + 16*(i_outer_inner_fetch_a + gid(0)*4))];
    barrier (CLK_LOCAL_MEM_FENCE) /* dependency: fetch_a */;
    for (int k_inner = 0; 15 - k_inner >= 0; ++k_inner)
    {
        acc[0] = fetch_b[lid(0)][k_inner]*fetch_a[k_inner][lid(1)][0] + acc[0];
        acc[1] = fetch_b[lid(0)][k_inner]*fetch_a[k_inner][lid(1)][1] + acc[1];
        acc[2] = fetch_b[lid(0)][k_inner]*fetch_a[k_inner][lid(1)][2] + acc[2];
        acc[3] = fetch_b[lid(0)][k_inner]*fetch_a[k_inner][lid(1)][3] + acc[3];
    }
    c[lid(0) + gid(1)*16 + 320*(lid(1) + 16*(gid(0)*4 + 0))] = acc[0];
    c[lid(0) + gid(1)*16 + 320*(lid(1) + 16*(gid(0)*4 + 1))] = acc[1];
    c[lid(0) + gid(1)*16 + 320*(lid(1) + 16*(gid(0)*4 + 2))] = acc[2];
    c[lid(0) + gid(1)*16 + 320*(lid(1) + 16*(gid(0)*4 + 3))] = acc[3];
}
```

Andreas Klöckner
Loo.py: A Loop Generation Tool
Example: Spectral Element 3D Laplacian

```

"[K] \rightarrow \{[i,j,k,e,m,o,gi] : 0<=i,j,k,m,o<%d and 0<=e<K and 0<=gi<6}\" % n,

"CSE: ur(i,j,k) = sum_float32(@o, D[i,o]*u[e,o,j,k])",
"CSE: us(i,j,k) = sum_float32(@o, D[j,o]*u[e,i,o,k])",
"CSE: ut(i,j,k) = sum_float32(@o, D[k,o]*u[e,i,j,o])",

"lap[e,i,j,k] = 
  sum_float32(m, D[m,i]*(G[0,e,m,j,k]*ur(m,j,k)
    + G[1,e,m,j,k]*us(m,j,k) + G[2,e,m,j,k]*ut(m,j,k)))
  + sum_float32(m, D[m,j]*(G[1,e,i,m,k]*ur(i,m,k)
    + G[3,e,i,m,k]*us(i,m,k) + G[4,e,i,m,k]*ut(i,m,k)))
  + sum_float32(m, D[m,k]*(G[2,e,i,j,m]*ur(i,j,m)
    + G[4,e,i,j,m]*us(i,j,m) + G[5,e,i,j,m]*ut(i,j,m)))",
```

Andreas Klöckner

Loo.py: A Loop Generation Tool
Capturing Variants

```python
knl = ...

def variant_cpu(knl):
    knl = lp.split_dimension(knl, "i", 16*4096, outer_tag="g.0", slabs=(0, 1))
    knl = lp.split_dimension(knl, "i_inner", 16,
                            inner_tag="unr")
    return knl

def variant_gpu(knl):
    knl = lp.split_dimension(knl, "i", 4*256, outer_tag="g.0", slabs=(0, 1))
    knl = lp.split_dimension(knl, "i_inner", block_size,
                            outer_tag="unr", inner_tag="l.0")
    return knl

for variant in [variant_cpu, variant_gpu]:
    kernel_gen = lp.generate_loop_schedules(variant(knl))
    # ...
```
Capturing Variants

```python
knl = ...

def variant_cpu(knl):
    knl = lp.split_dimension(knl, "i", 16*4096, outer_tag="g.0", slabs=(0, 1))
    knl = lp.split_dimension(knl, "i_inner", 16,
                             inner_tag="unr")
    return knl

def variant_gpu(knl):
    knl = lp.split_dimension(knl, "i", 4*256, outer_tag="g.0", slabs=(0, 1))
    knl = lp.split_dimension(knl, "i_inner", block_size,
                             outer_tag="unr", inner_tag="l.0")
    return knl

for variant in [variant_cpu, variant_gpu]:
    kernel_gen = lp.generate_loop_schedules(variant(knl))
    # ...
```

Easy to *non-redundantly* capture multiple variants of the same kernel.
Free extras:

- A-priori bounds checking (future)
- Generate a sequential version of the code
- Automatic Benchmarking
- Free tuning advice
  - Local memory layout
  - Suboptimal use of hw parallelism
  - Based on knowledge about target hardware
- Automatic Testing
  - ... against sequential version
  - ... which is easier to verify
Summing up

- Two modes of operation:
  - Standalone—write spec, run loopy, copy’n’paste generated code
  - In-process use, direct integration with PyOpenCL
- License: will be MIT
  - once documentation and paper are written
- Mostly feature-complete, early beta.
- Flat data structure:
  - Easy to manipulate
  - Kernel fusion
- Future: non-convex domains (→ stencils)
Outline

1. Introduction

2. A Different Way of Writing Compute Codes

3. Code Generation, Generally

4. Conclusions
   - Conclusions
Outline

1 Introduction

2 A Different Way of Writing Compute Codes

3 Code Generation, Generally

4 Conclusions
   - Conclusions
GPU DG Showcase

Eletromagnetism
GPU DG Showcase

Eletromagnetism

Poisson
GPU DG Showcase

Eletromagnetism

CFD

Loo.py: A Loop Generation Tool
GPU DG Showcase

Eletromagnetism

CFD
Where to from here?

PyCUDA, PyOpenCL, hedge

→ http://www.cims.nyu.edu/~kloeckner/

GPU-DG Article


Also: Intro in GPU Computing Gems Vol 2

GPU RTCG

GPUs and scripting work surprisingly well together

- Enable Run-Time Code Generation

GPU-DG is significantly faster than CPU-DG

- Method well-suited a priori
- Numerous tricks enable good performance
- But who likes coding by hand?

So: Distill lessons learned into a tool

- *Not* a general tool
- It’s good at very specific things
- Builds on existing tools, experience

Hopes for loopy:

- General enough to be broadly useful
- Evolve over time: generality, performance, ...
Questions?

Thank you for your attention!

http://www.cims.nyu.edu/~kloeckner/
Image Credits

- ?/! Marks: sxc.hu/svilen001
- OpenCL logo: The Khronos Group, Inc.
- Python logo: Python Software Foundation
- CUDA Logo: Nvidia Corporation
- C870 GPU: Nvidia Corp.
- Glass dollar: sxc.hu/flaivoloka